

#2

IN THE UNITED STATES PATENT OFFICE

J1040 U.S. PRO
10/020656
10/29/01

Applicant: Komatsu et al
Serial No.: unassigned
Filed: herewith

Date: October 29, 2001
Docket No.: JP920000285
Group Art Unit: unassigned
Examiner: unassigned

For: Program Optimization

SUBMISSION OF CERTIFIED PRIORITY DOCUMENT

To the Assistant Commissioner of Patents:

Enclosed herewith is a certified copy of the above-identified Application No. 2000-331427, filed in Japan on October 30, 2000, for which applicant claims priority under 35 U.S.C. & 119.

It is requested that this be made part of record in the subject application.

Respectfully submitted,

By: Lauren Bruzzone
Lauren Bruzzone
(Reg No 35,082)
Telephone: (914) 945-3255

IBM CORPORATION
Intellectual Property Law Dept.
T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, N.Y. 10598
(914) 945-3255

ExpMailCert: EV001688011US
Date of Deposit: Oct. 29, 2001

日本国特許庁
JAPAN PATENT OFFICE

J1040 U.S. PRO
10/020656
10/29/01

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed with this Office

出願年月日
Date of Application:

2000年10月30日

出願番号
Application Number:

特願2000-331427

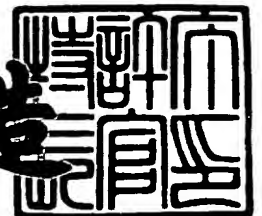
出願人
Applicant(s):

インターナショナル・ビジネス・マシーンズ・コーポレーション

2001年 6月19日

特許庁長官
Commissioner,
Japan Patent Office

及川耕造



出証番号 出証特2001-3057751

【書類名】 特許願

【整理番号】 JP9000285

【提出日】 平成12年10月30日

【あて先】 特許庁長官 殿

【国際特許分類】 G06F 5/06

【発明者】

 【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

 【氏名】 稲垣 達氏

【発明者】

 【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

 【氏名】 小松 秀昭

【特許出願人】

 【識別番号】 390009531

 【氏名又は名称】 インターナショナル・ビジネス・マシーンズ・コーポレーション

【代理人】

 【識別番号】 100086243

 【弁理士】

 【氏名又は名称】 坂口 博

【代理人】

 【識別番号】 100091568

 【弁理士】

 【氏名又は名称】 市位 嘉宏

【代理人】

 【識別番号】 100106699

 【弁理士】

 【氏名又は名称】 渡部 弘道

【復代理人】

【識別番号】 100104880

【弁理士】

【氏名又は名称】 古部 次郎

【選任した復代理人】

【識別番号】 100100077

【弁理士】

【氏名又は名称】 大場 充

【手数料の表示】

【予納台帳番号】 081504

【納付金額】 21,000円

【提出物件の目録】

【物件名】 明細書 1

【物件名】 図面 1

【物件名】 要約書 1

【包括委任状番号】 9706050

【包括委任状番号】 9704733

【包括委任状番号】 0004480

【プルーフの要否】 要

【書類名】 明細書

【発明の名称】 プログラムの最適化方法及びこれを用いたコンパイラ

【特許請求の範囲】

【請求項 1】 プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行う最適化方法において、

処理対象である前記プログラムを解析し、例外が発生する可能性のある例外発生可能命令と、当該例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出するステップと、

検出された前記例外発生検出命令を、前記例外の発生条件を検出する第 1 の命令と、前記例外の発生条件が検出された場合に処理を前記例外処理へ条件分岐させる第 2 の命令とに分割するステップと、

前記第 1 の命令に対して、前記例外の発生条件を検出した場合に前記第 2 の命令に移行し、前記例外発生条件を検出しなかった場合に前記例外発生可能命令に移行するように依存関係を設定するステップと
を含むことを特徴とするプログラムの最適化方法。

【請求項 2】 前記命令の依存関係を設定するステップの後に、

プログラムの所定の範囲における複数の前記例外発生検出命令に基づいて得られた複数の前記第 2 の命令をまとめるステップをさらに含み、

複数の前記例外発生検出命令に基づいて得られた複数の前記第 1 の命令が例外の発生条件を検出したことを一括して判断する、

請求項 1 に記載のプログラムの最適化方法。

【請求項 3】 前記命令を分割するステップは、前記第 1 の命令として、前記例外の発生条件を検出した場合にフラグを立てる命令を生成するステップを含み、

前記第 2 の命令をまとめるステップは、複数の前記例外発生検出命令に基づいて得られた複数の前記第 1 の命令において、少なくともいずれかが前記フラグを立てている場合に、例外が発生したことを検出し、前記第 2 の命令に処理を移行させる命令を生成するステップを含む、

請求項 2 に記載のプログラムの最適化方法。

【請求項 4】 前記例外発生可能命令を検出するステップの後に、

前記例外発生可能命令が副作用を伴う場合、前記例外発生検出命令の分割前における当該副作用に関する順序制約を保存する補償コードを生成するステップをさらに含む、

請求項 1 に記載のプログラムの最適化方法。

【請求項 5】 前記命令の依存関係を設定するステップの後に、

前記第 1 の命令を条件分岐とし、当該条件分岐を反映するようにプレディケートの割付を行うステップをさらに含む、

請求項 1 に記載のプログラムの最適化方法。

【請求項 6】 前記命令の依存関係を設定するステップの後に、

前記第 1 の命令を条件分岐とし、コードスケジューリングの結果に応じて、前記第 1 の命令の分岐先に、前記第 1、第 2 の命令に分割される前の前記例外発生検出命令に関する順序制約を満足するように補償コードを生成するステップをさらに含む、

請求項 1 に記載のプログラムの最適化方法。

【請求項 7】 プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行うコンパイラにおいて、

処理対象である前記プログラムを解析し、当該プログラム中の演算子の依存関係を示すグラフを生成するグラフ生成部と、

生成された前記グラフを編集し、例外による前記演算子の順序制約を緩和するグラフ編集部と、

編集された前記グラフにおける前記演算子の依存関係を反映させたプログラムコードを生成するコード再生部とを備え、

前記グラフ編集部は、

前記グラフ中から、例外が発生する可能性のある例外発生可能命令と、当該例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出し、

検出された前記例外発生検出命令を、前記例外の発生条件を検出する第 1 の命令と、前記例外の発生条件が検出された場合に処理を前記例外処理へ条件分岐さ

せる第 2 の命令とに分割し、

前記グラフ上の前記第 1 の命令に対して、前記例外の発生条件を検出した場合に前記第 2 の命令に移行し、前記例外発生条件を検出なかった場合に前記例外発生可能命令に移行するように依存関係を設定すること
を特徴とするコンパイラ。

【請求項 8】 前記グラフ編集部は、

前記グラフから、前記例外発生検出命令の分割前における前記例外発生可能命令に関する順序制約と、前記例外発生検出命令の分割前における当該例外発生検出命令に先行する順序制約とを取り除く、
請求項 7 に記載のコンパイラ。

【請求項 9】 前記グラフ編集部は、

プログラムの所定の範囲における前記グラフ上の複数の前記例外発生検出命令に基づいて得られた複数の前記第 2 の命令を合成する、
請求項 7 に記載のコンパイラ。

【請求項 10】 前記グラフ編集部は、

前記例外発生可能命令が副作用を伴う場合に、前記例外発生検出命令の分割前における当該副作用に関する順序制約を保存するように、前記グラフ上の前記例外発生可能命令に対して順序制約を設定する、
請求項 7 に記載のコンパイラ。

【請求項 11】 前記グラフ編集部は、

前記第 1 の命令を条件分岐とし、コードスケジューリングの結果に応じて、前記グラフにおける前記第 1 の命令の分岐先に、前記第 1、第 2 の命令に分割される前の前記例外発生検出命令に関する順序制約を満足するように補償コードを生成する、
請求項 7 に記載のコンパイラ。

【請求項 12】 プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行うコンパイラにおいて、

前記ソースコードを編集可能な中間コードに変換する中間コード生成部と、
前記中間コードに対して最適化を行う最適化部と、

最適化された前記中間コードから機械語コードを生成する機械語コード生成部とを備え、

前記最適化部は、

処理対象である前記プログラムの中間コードを解析し、当該プログラムの所定の範囲において、例外が発生する可能性のある例外発生可能命令を他の命令に先んじて実行するように、当該プログラムを変形し、

前記例外発生可能命令において例外が発生した場合に当該例外発生可能命令以降の命令が実行されないように、前記プログラムにおける命令間の依存関係を設定し、

前記プログラムの所定の範囲において、複数存在する前記例外発生可能命令のうち少なくとも一つが例外が発生する場合に、当該例外の発生を検出し、当該例外に対応する例外処理へ移行するように、前記プログラムを変形することを特徴とするコンパイラ。

【請求項 1 3】 前記最適化部は、

前記例外発生可能命令が副作用を伴う場合に、前記プログラムの変形前における当該副作用に関する順序制約を保存するように、前記グラフ上の前記例外発生可能命令に対して順序制約を設定する、

請求項 1 2 に記載のコンパイラ。

【請求項 1 4】 プログラムのソースコードを入力する入力装置と、

入力された前記プログラムをコンパイルして機械語コードに変換するコンパイラと、

機械語コードに変換された前記プログラムを実行する処理装置とを備えたコンピュータ装置において、

前記コンパイラは、

処理対象である前記プログラムを解析し、例外が発生する可能性のある例外発生可能命令と、当該例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出し、

検出された前記例外発生検出命令を、前記例外の発生条件を検出する第 1 の命令と、前記例外の発生条件が検出された場合に処理を前記例外処理へ条件分岐さ

せる第 2 の命令とに分割し、

前記第 1 の命令に対して、前記例外の発生条件を検出した場合に前記第 2 の命令に移行し、前記例外発生条件を検出なかった場合に前記例外発生可能命令に移行するように依存関係を設定すること
を特徴とするコンピュータ装置。

【請求項 1 5】 前記コンパイラは、

プログラムの所定の範囲における複数の前記例外発生検出命令に基づいて得られた複数の前記第 1 の命令が例外の発生条件を検出したことを一括して判断するように、複数の当該例外発生検出命令に基づいて得られた複数の前記第 2 の命令をまとめる、

請求項 1 4 に記載のコンピュータ装置。

【請求項 1 6】 コンピュータに実行させるプログラムを当該コンピュータの入力手段が読取可能に記憶した記憶媒体において、

前記プログラムは、

処理対象であるプログラムを解析し、例外が発生する可能性のある例外発生可能命令と、当該例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出する処理と、

検出された前記例外発生検出命令を、前記例外の発生条件を検出する第 1 の命令と、前記例外の発生条件が検出された場合に処理を前記例外処理へ条件分岐させる第 2 の命令とに分割する処理と、

前記第 1 の命令に対して、前記例外の発生条件を検出した場合に前記第 2 の命令に移行し、前記例外発生条件を検出なかった場合に前記例外発生可能命令に移行するように依存関係を設定する処理と

を前記コンピュータに実行させる記憶媒体。

【請求項 1 7】 前記記憶媒体に記憶される前記プログラムは、前記命令の依存関係を設定する処理の後に、

プログラムの所定の範囲における複数の前記例外発生検出命令に基づいて得られた複数の前記第 2 の命令をまとめる処理をさらに含み、

複数の前記例外発生検出命令に基づいて得られた複数の前記第 1 の命令が例外

の発生条件を検出したことを一括して判断する、
請求項 1 6 に記載の記憶媒体。

【請求項 1 8】 コンピュータに、

処理対象であるプログラムを解析し、例外が発生する可能性のある例外発生可能命令と、当該例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出する処理と、検出された前記例外発生検出命令を、前記例外の発生条件を検出する第 1 の命令と、前記例外の発生条件が検出された場合に処理を前記例外処理へ条件分岐させる第 2 の命令とに分割する処理と、前記第 1 の命令に対して、前記例外の発生条件を検出した場合に前記第 2 の命令に移行し、前記例外発生条件を検出なかった場合に前記例外発生可能命令に移行するように依存関係を設定する処理とを実行させるプログラムを記憶する記憶手段と、

前記記憶手段から前記プログラムを読み出して当該プログラムを送信する送信手段とを備えたことを特徴とするプログラム伝送装置。

【請求項 1 9】 前記記憶手段に記憶される前記プログラムは、前記命令の依存関係を設定する処理の後に、

プログラムの所定の範囲における複数の前記例外発生検出命令に基づいて得られた複数の前記第 2 の命令をまとめる処理をさらに含み、

複数の前記例外発生検出命令に基づいて得られた複数の前記第 1 の命令が例外の発生条件を検出したことを一括して判断する、
請求項 1 8 に記載のプログラム伝送装置。

【発明の詳細な説明】

【0 0 0 1】

【発明の属する技術分野】

本発明は、コンピュータプログラムの最適化方法に関し、特に例外を起こす可能性のある命令（以下、例外発生可能命令と称す）を含むプログラムに対して命令レベルでの並列性を得るための最適化方法に関する。

【0 0 0 2】

【従来の技術】

通常、プログラミング言語で記述されたプログラムのソースコードをコンパイルする際、コンピュータにおける実行速度の向上を図るために、当該プログラムの最適化を行っている。

最適化の手法には種々の方法があるが、米国インテル社及び米国ヒューレット・パッカード社による I A - 6 4 のアーキテクチャに対応した C P U のように、命令レベルの並列実行が可能なプロセッサ上で実行されることを前提として、命令レベルでの並列性を抽出するためのコードスケジューリングにおいて、できるだけ並行して発行可能な命令が多くなるように、命令の順番の入れ替えを行うという最適化がある。

【 0 0 0 3 】

ところがプログラム中には、順番の入れ替えに対する様々な制約があり、コードスケジューリングの働きは抑えられている。順番の入れ替えに対する制約には、コントロールによる制約、メモリアクセスによる制約、例外による制約がある。これらの制約のうち、コントロールによる制約とメモリアクセスによる制約に対しては、ハードウェアサポートによって緩和する機構が実用化されている。しかし、例外による制約をハードウェアサポートによって緩和する機構は実現されていない。

【 0 0 0 4 】

また、プログラムにおける命令レベルの並列性を抽出した場合、先行的に発行された命令が起こした例外を処理することが必要である。従来この種の処理の代表的な方法は、そのような命令が上げた例外を即時には実行せず、最適化を行う前のプログラムにおいてその命令がもともと置かれていた場所で例外発生の有無を判定し、実際の例外処理を行う方法である。これによって、例外発生時にもオリジナルのコンテキストを保存しながら、例外発生可能命令を先行的に実行できる。

【 0 0 0 5 】

ところで、例外が発生する可能性のある命令を投機的に実行して並列性を抽出するため、従来技術は、投機的に実行される例外発生可能命令で起きる例外（以下、このような例外を投機的例外と称す）を抑制するハードウェアサポートを

仮定していた。

この種の従来技術としては、例えば下記の文献 1 に開示された技術がある。

文献 1 : Scott A. Mahlke, William Y. Chen, Roger A. Bringmann, Richard E. Hank, Wen-Mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker, "Sentinel Scheduling: A Model for Compiler-Controlled Speculative Execution", ACM Transactions on Computer Systems, Vol. 11, No. 4, November 1993, Pages 376-408

【 0 0 0 6 】

文献 1 は、ハードウェアサポートの元に、コンパイラが投機的な命令移動と例外発生時の補償コードの生成を行う方法を提案している。ここで仮定しているハードウェアでは、投機的な命令に関しては例外を上げる代わりに特別な演算結果を生成し、その値が後続の投機的な命令によって推移的に伝搬される。そして、投機的でない命令はこの特別な値を使用したときに例外を発生する。実際に例外が起きたかどうかを確認する命令はsentinel instructionと呼ばれる。

【 0 0 0 7 】

この従来技術では、以下の手順で例外発生可能命令を投機的にスケジューリングする。

1. 各例外発生可能命令について、生成される値を最後に使用する命令(potential sentinel instruction)を求める。
2. リストスケジューリングによって、各命令の最も早い実行開始時刻を求める。例外発生可能命令が分岐、副作用(メモリへの書き込み)、他の例外発生可能命令を越えてスケジューリングされた場合、その命令は投機的である。したがって、手順 1 で求めた情報を使って、必要な場合は明示的なsentinel instructionを生成する。
3. 例外状態からの回復をソフトウェアで行う場合は、各sentinel instructionに対応した例がハンドラに対して例外発生時の補償コードを生成する。

【 0 0 0 8 】

また、この種の他の従来技術として、特開平 1 2 - 0 2 0 3 2 0 号公報に開示

された技術がある。同公報は、コンパイラおよびランタイムライブラリのサポートによって例外発生可能命令の投機的な実行を実現する手法を開示している。

【 0 0 0 9 】

この従来技術では、以下の手順で例外発生可能命令の投機的実行を行う。

1. コンパイラが、最終的に生成されるオブジェクトコード中で、例外発生可能命令の投機的実行によって実行順序が元のプログラムと変わった部分を割り込み禁止区間として登録する。
2. 例外発生時に、例外ハンドラが割り込み禁止区間情報を元にして、補償コードを動的に生成し実行する。

【 0 0 1 0 】

【発明が解決しようとする課題】

上述したように、プログラムを命令レベルで並列実行させるためのコードスケジューリングには種々の制約があり、特に例外による制約ハードウェアサポートによって緩和する機構は実現されていない。

これに対し、J a v aのような、例外の取り扱いが厳密でなければならない言語においては、例外発生可能命令間の制約が、プログラムにおける命令レベルの並列性の抽出に及ぼす影響が大きい。このため、J a v aなどの例外の取り扱いが厳密でなければならない言語で記述されたプログラムに対するコンパイラにおいて、命令レベルの並列性を抽出することは困難であった。

【 0 0 1 1 】

J a v a言語を例としてさらに詳細に説明する。

J a v a言語の仕様は、プログラムの実行中に発生する例外条件の厳密な取り扱いを定めている。システムで定められている例外状態（オブジェクトのポインタがn u l lである場合、配列の長さを越えた添字でアクセスした場合、整数の0除算を行った場合など）及びユーザーが定義した例外状態は、全て例外の発生を実行時に検出して、ユーザーにより定義された例外ハンドラで処理することができる。そして、例外ハンドラの実行が終了すると制御は例外を検出した領域から脱出する。

【 0 0 1 2 】

ここで、例外の発生は制御フローを変更するので、プログラム中の副作用（配列の要素やクラス、オブジェクトのフィールドへの書き込み）の発生と例外の発生の間には全順序がある。すなわち、ユーザー定義の例外ハンドラが実行される時点では、プログラム中で次の二つの条件を満足しなければならない。

1. 例外の発生より先に行われた副作用は全て観測されなければならない。
2. 例外の発生より後に行われる副作用は観測されてはならない。

【 0 0 1 3 】

J a v a では、配列の要素やオブジェクトのフィールドへの読み出し及び書き込みなどの基本的な操作が実行時の例外を伴う。すなわち、J a v a プログラムが機械語命令にコンパイルされると、これらの操作は実行時の例外を検出する命令と、実際の操作を行うメモリアクセスや演算に変換される。そして、メモリアクセスや演算は例外を発生する可能性がある。

ここで、発生する例外は、アクセス違反やゼロで割る除算などを実行することによってプロセッサ（ハードウェア）が取る状態としての例外である。プログラムの実行に伴う例外としては、この他、プログラム言語の仕様外の入力に対するプログラムの実行状態としての例外がある。以下の説明において、特に前者の例外を他の例外と区別する必要がある場合は、ハードウェア例外と称する。

ハードウェア例外は、機械語プログラムの間違いによって、プロセッサが不正なメモリ領域に読み書きを行ったり、計算できない演算を行ったりしたために、正しく実行を継続できない場合に発生する。一方、その他の例外は、ユーザーの書いたプログラムの間違いにより、配列のサイズを超えた要素を読み書きしたり関数に正しくない入力値を与えたりした場合のエラー処理として、言語及びプログラムが定める例外である。

【 0 0 1 4 】

このハードウェア例外の発生を回避するため、コードスケジューリングにおいて、副作用を行う命令、例外発生を検出する命令、例外発生可能命令の間に実行順序の制約が生じる。

J a v a プログラムにおいては、この実行順序の制約を伴う配列要素のアクセスやフィールド変数のアクセスが頻繁に現れるため、これらの順序制約が、コー

ドスケジューリングにおいて命令レベルの並列性を抽出する処理の大きな妨げになっていた。

【 0 0 1 5 】

また、上述したように、プログラムにおける命令レベルの並列性を抽出した場合、先行的に発行された命令が起こした例外を処理することが必要であるが、従来は、そのような命令が上げた例外を即時には実行せず、最適化を行う前のプログラムにおいてその命令がもともと置かれていた場所で例外発生の有無を判定し、実際の例外処理を行うことにより対応していた。

【 0 0 1 6 】

しかしこの場合、先行的に上げられた例外の発生の有無を判定する命令のコストを無視することはできない。例えば上述した I A - 6 4 では、投機的例外を検出するための特別な機械語が用意されているが、大規模に命令の移動を行った場合、そのような例外の発生を判定するための命令が実行速度に与える影響は大きくなるものと考えられる。

【 0 0 1 7 】

文献 1 に記載された従来技術では、ハードウェア例外を発生する可能性のある命令を投機的に実行することはできるが、副作用や例外発生を検出する命令の間には順序制約が残る。副作用のある命令を投機的に実行するためには、さらにメモリへの書き込みをキャンセルするためのハードウェアコストが必要である。

また、この従来技術は、ハードウェアサポートを必要とするため、投機的実行を行うハードウェアを持たない一般のプロセッサには直接適用できない。

さらに、ソフトウェアによりこの従来技術による方法をエミュレーションする手法は付加的な命令によるコストが高く現実的ではない。

したがって、この従来技術は、単純なハードウェア例外発生可能命令の投機実行と、例外状態の回復によるモデルではオーバーヘッドが高く、J a v a 言語のコンパイルにおける例外発生可能命令の順序制約を緩和するには十分ではないと言える。

【 0 0 1 8 】

また、特開平 1 2 - 0 2 0 3 2 0 号公報に記載された従来技術では、例外が発

生しないときにはソフトウェアオーバーヘッドなしで実現できる。しかし、補償コードを例外ハンドラで実行時に生成するため、機械語命令列が全て決定された後の最適化、すなわちレジスタ割付後のコードスケジューリングにおける最適化にしか適用できない。このため、コード変形を伴う最適化で、例外発生可能命令の投機的な実行による並列性を利用することができない。

【 0 0 1 9 】

そこで、本発明は、例外発生可能命令の他の命令への先行制約をソフトウェア的に緩和することによって、例外発生可能命令を含むプログラムの命令レベル並列性を効果的に得ることを目的とする。

【 0 0 2 0 】

【課題を解決するための手段】

かかる目的のもと、本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行う最適化方法において、処理対象のプログラムを解析し、例外発生可能命令と、例外発生検出命令とを検出するステップと、検出された例外発生検出命令を、例外の発生条件を検出する第1の命令と、例外の発生条件が検出された場合に処理を例外処理へ条件分岐させる第2の命令とに分割するステップと、第1の命令に対して、例外の発生条件を検出した場合に第2の命令に移行し、例外発生条件を検出しなかった場合に例外発生可能命令に移行するように依存関係を設定するステップとを含むことを特徴とする。

このようにプログラムを変形し、依存関係を設定することによって、例外発生可能命令における順序制約を緩和することができる。

【 0 0 2 1 】

ここで、この最適化方法は、命令の依存関係を設定するステップの後に、プログラムの所定の範囲における複数の例外発生検出命令に基づいて得られた複数の第2の命令をまとめるステップをさらに含む。これにより、複数の例外発生検出命令に基づいて得られた複数の第1の命令が例外の発生条件を検出したことを一括して代表的な判断を行う。

【 0 0 2 2 】

さらに詳しくは、この最適化方法において、命令を分割するステップは、第1の命令として、例外の発生条件を検出した場合にフラグを立てる命令を生成するステップを含み、第2の命令をまとめるステップは、複数の例外発生検出命令に基づいて得られた複数の第1の命令において、少なくともいずれかがフラグを立てている場合に、例外が発生したことを検出し、第2の命令に処理を移行させる命令を生成するステップを含む。

上述したプログラムの所定の範囲として基本ブロックを用いることができる。そして、この基本ブロック内における例外の発生の判定を一つのフラグを用いてまとめて行うことにより、実際の例外発生を検出するソフトウェアオーバーヘッドを一つの分岐命令で済ませることができる。

【0023】

また、この最適化方法は、例外発生可能命令を検出するステップの後に、例外発生可能命令が副作用を伴う場合、例外発生検出命令の分割前におけるこの副作用に関する順序制約を保存する補償コードを生成するステップをさらに含む。

ここで、副作用とは、命令の実行に伴って行われる所定のデータのメモリへの書き込みである。

【0024】

さらに、この最適化方法は、コンパイル後のプログラムを実行するコンピュータ装置のCPUがプレディケート付き命令を実行する機能を持つ場合、例外発生可能命令をガードするため、命令の依存関係を設定するステップの後に、第1の命令を条件分岐とし、この条件分岐を反映するようにプレディケートの割付を行うステップをさらに含む。

【0025】

これに対し、この最適化方法は、コンパイル後のプログラムを実行するコンピュータ装置のCPUがプレディケート付き命令を実行する機能を持たない場合、例外発生可能命令をガードするため、命令の依存関係を設定するステップの後に、第1の命令を条件分岐とし、コードスケジューリングの結果に応じて、第1の命令の分岐先に、第1、第2の命令に分割される前の例外発生検出命令に関する順序制約を満足するように補償コードを生成するステップをさらに含む。

【 0 0 2 6 】

また本発明は、上述した最適化方法を含むコンパイルを、コンピュータ装置に実行させるコンピュータプログラムとして作成し、このコンピュータプログラムを格納した記憶媒体や、このコンピュータプログラムを伝送する伝送装置として提供することができる。

さらに本発明は、このコンピュータプログラムを実行するコンパイラを備えたコンピュータ装置として提供することができる。

【 0 0 2 7 】

また、本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行うコンパイラにおいて、処理対象のプログラムを解析し、このプログラム中の演算子の依存関係を示すグラフを生成するグラフ生成部と、生成されたこのグラフを編集し、例外による演算子の順序制約を緩和するグラフ編集部と、編集されたこのグラフにおける演算子の依存関係を反映させたプログラムコードを生成するコード再生部とを備え、グラフ編集部は、このグラフ中から、例外が発生する可能性のある例外発生可能命令と、この例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出し、検出された例外発生検出命令を、この例外の発生条件を検出する第1の命令と、この例外の発生条件が検出された場合に処理を例外処理へ条件分岐させる第2の命令とに分割し、このグラフ上の第1の命令に対して、例外の発生条件を検出した場合に第2の命令に移行し、例外発生条件を検出しなかった場合に例外発生可能命令に移行するように依存関係を設定することを特徴とする。

【 0 0 2 8 】

ここで、このグラフ編集部は、このグラフから、例外発生検出命令の分割前における例外発生可能命令に関する順序制約と、例外発生検出命令の分割前におけるこの例外発生検出命令に先行する順序制約とを取り除く。

これにより、例外発生可能命令に対する順序制約をプログラムの所定の領域（基本ブロックなど）におけるクリティカルパス上から除去することができる。

【 0 0 2 9 】

また、本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行うコンパイラにおいて、このソースコードを編集可能な中間コードに変換する中間コード生成部と、この中間コードに対して最適化を行う最適化部と、最適化された中間コードから機械語コードを生成する機械語コード生成部とを備え、この最適化部は、処理対象であるプログラムの中間コードを解析し、プログラムの所定の範囲において、例外が発生する可能性のある例外発生可能命令を他の命令に先んじて実行するように、このプログラムを変形し、この例外発生可能命令において例外が発生した場合にこの例外発生可能命令以降の命令が実行されないように、このプログラムにおける命令間の依存関係を設定し、このプログラムの所定の範囲において、複数存在する例外発生可能命令のうち少なくとも一つが例外が発生する場合に、この例外の発生を検出し、この例外に対応する例外処理へ移行するように、このプログラムを変形することを特徴とする。

【 0 0 3 0 】

さらにここで、この最適化部は、この例外発生可能命令が副作用を伴う場合に、このプログラムの変形前におけるこの副作用に関する順序制約を保存するように、このグラフ上の例外発生可能命令に対して順序制約を設定する。

【 0 0 3 1 】

【発明の実施の形態】

以下、添付図面に示す実施の形態に基づいて、この発明を詳細に説明する。

まず、本発明の概要について説明する。本発明は、例外を起こす可能性のある命令（例外発生可能命令）における他の命令に対する先行制約をソフトウェア的に緩和することによって、例外発生可能命令を含むプログラムの命令レベル並列性を得る。

【 0 0 3 2 】

本発明において、コンパイラは、他の命令に先んじて発行された例外発生可能命令がソフトウェアによってガードされるようにコードの変形を行う。そして、投機的に実行される例外発生可能命令により例外（投機的例外）が起きる場合、コンパイラが用意した例外ハンドラに分岐し、必要な副作用（メモリへの書き込

み)及び例外の検出を行ってから例外の発生を表すフラグを立て、ガードされていない命令のコンテキストに復帰して実行を再開する。このように、コンパイラにおいて例外の発生をコントロールすることにより、より多くの種類の命令を先行的に発行できる。

その後、例外発生可能命令が元々配置されていたコンテキストにおいて、上記の例外の発生を表すフラグを判定し、フラグが立っていれば、当該例外に対する固有の例外処理を行う。

このように、投機的例外の判定を条件分岐やプレディケートなどのソフトウェアで行うことにより、例外発生可能命令は、理論的に、コントロールによる制約、メモリアクセスによる制約、例外による制約の全てに制限されずに先行的に発行できるようになる。

【 0 0 3 3 】

しかしながら、このような投機的例外を判定するための命令を実行するコストは無視できるものではない。さらに、ハードウェアでサポートされていない投機処理を行う場合は、この例外判定をソフトウェアによって行うこととなるため、そのコストは大きくなる。

【 0 0 3 4 】

そこで、本発明は、プログラムにおける基本ブロックの先頭において、その基本ブロック内におけるフラグのいずれか一つでも立っていれば、これを一括して検出する代表的な判定を用意する。この判定は、フラグをビットワイズに表現することによって、機械語 1 命令で高速に処理できる。

この代表的な判定において、基本ブロック内のいずれかのフラグが立っていると判明した場合、まず、オリジナルのプログラムで記述されているコンテキストを回復させるのに必要十分な命令を実行する。そして、立っているフラグを検出し、当該フラグに対応する例外に対する処理を行う。

なお、基本ブロックとは、ストレートコード、すなわちコントロールフローが途中に入ることもなく、途中から出ることもないようなコード列の範囲をブロックで示したものである。

【 0 0 3 5 】

図 1 は、本発明の実施の形態におけるコンパイラの構成を説明する図である。

本実施の形態では、処理対象である、例外の取り扱いが厳密でなければならない言語として J a v a を用いる例について説明する。すなわち、最適化の対象であるプログラムを J a v a プログラムとし、コンパイラを J a v a の Just In Time Compiler とする。したがって、図 1 に示す本実施の形態のコンパイラ 1 0 0 は、J a v a バイトコードの形式で記述されたプログラムを入力してコンパイルし、当該プログラムを実行する計算機に対応した機械語コード形式に変換して出力する。ただし、本発明を、他の種々のプログラム言語で記述されたプログラムに対するコンパイラに適用できることは言うまでもない。

また、コンパイラ 1 0 0 として、J a v a の Just In Time Compiler を想定する場合、コンパイラ 1 0 0 は、当該プログラムを実行するコンピュータ装置に搭載されることとなる。すなわち、図示しないがこのコンピュータ装置は、プログラムのソースコードを入力する入力装置と、コンパイラ 1 0 0 を実現するコンピュータプログラムを格納したメモリと、メモリに格納されたコンピュータプログラムの制御によりコンパイラ 1 0 0 としての処理を実行すると共に、コンパイラ 1 0 0 により機械語コードに変換されたプログラムを実行する計算機（C P U）とを備える。また、このコンピュータ装置は、コンパイラ 1 0 0 を実現するコンピュータプログラムを、上述した C D - R O M や フロッピーディスクから読み出すディスクドライブや、ネットワークを介して受信する受信部を備える。

【 0 0 3 6 】

図 1 を参照すると、本実施の形態のコンパイラ 1 0 0 は、四つ組中間コード生成部 1 0 と、四つ組中間コード最適化部 2 0 と、機械語コード生成部 3 0 とを備える。

上記構成において、四つ組中間コード生成部 1 0 は、J a v a バイトコード形式で与えられたプログラムを解析し、四つ組形式で表現された中間コード（以下、四つ組中間コードと称す）に変換する。

四つ組中間コード最適化部 2 0 は、四つ組中間コード生成部 1 0 により生成された四つ組中間コードに対して、すなわち、最終的に生成される機械語コードの実行時間が短くなるように、計算の冗長性を取り除いたり、中間コードを移動し

たりする。この最適化の過程で、プログラム中の例外発生可能命令を投機的に実行するように命令の順序の入れ替えを行う。

機械語コード生成部 3 0 は、最適化された四つ組中間コードを、当該プログラムを実行する計算機に対応した機械語コード形式に変換し出力する。

【 0 0 3 7 】

本実施の形態は、J a v a 言語の例外処理をコンパイルする際の以下のような特徴に注目して、ソフトウェアによる例外発生可能命令の投機実行を効率的に行う。

1. 例外処理は必ず制御ブロックからの脱出である。また、ユーザー定義の例外ハンドラで検出できるのは副作用の結果だけであり、それ以外の中間結果は例外ハンドラ以降の実行では参照されない。したがって、投機的に実行した例外発生可能命令が例外を起す場合は、正確に全ての状態を回復する必要はなく、副作用の順序を保証し、例外の種類と引数を特定できれば良い。

2. 例外はプログラムにおける「例外的条件」を扱うので、プログラムの実行全体から見ると例外が発生する頻度は比較的低いと仮定できる。したがって、例外が発生した場合の処理のソフトウェアオーバーヘッドはある程度許容できる。

3. 本発明は、J a v a チップなどではない通常のプロセッサを対象としてコンパイルを行う場合、例外の検出のためにソフトウェアによる明示的な命令列によるオーバーヘッドを伴っている。代表的かつ最も頻繁に現れる例外は、配列の長さを超えたインデックスによるアクセスを検出する例外 (ArrayIndexOutOfBoundsException) である。そこで、配列サイズとインデックス変数をチェックする既存の命令を使って、投機的例外の発生を検出することができる。

【 0 0 3 8 】

J a v a 言語における以上の特徴を利用して、J a v a 言語の例外発生可能命令をソフトウェアで投機的に実行するために以下の基本方針を採用する。

1. J a v a 言語における例外を検出する既存の命令列を利用して、ハードウェア例外発生可能命令の実行を、条件分岐やプレディケート付き命令によってソフトウェア的にガードする。すなわち、ハードウェア例外が発生する場合には当該例外発生可能命令は実行されない。これによって、例外が発生しない場合の実行

において付加的な命令オーバーヘッドを導入しなくて済むこととなる。ガードされた例外発生可能命令は、ガードに対して制御依存を持つ代わりに副作用や他の例外発生可能命令に対して順序制約を持たなくなるので、例外発生可能命令による順序制約が緩和される。

2. J a v a 言語の定める例外が発生したかどうかを検出し、ユーザー定義の例外ハンドラに分岐する命令と例外発生条件を検出する命令とを分離し、基本ブロック内で例外が起ったことを表す代表のフラグを用いる。これによって、実際に例外ハンドラに条件分岐する処理を基本ブロック内で一つにまとめることができるので、実際の例外発生を検出するソフトウェアオーバーヘッドを一つの分岐命令で済ませることができる。

3. 例外発生検出時の補償コードを、例外発生検出命令の分岐先（または、プレディケート付き命令における負の条件のプレディケートによる分岐先）に生成する。これによって補償コードのための余分な分岐命令を通常実行されるパスに挿入することを回避することができる。例外処理は必ず脱出であるため、補償コードは副作用及び例外の検出によって構成される。補償コードを生成するために、副作用や例外発生可能命令の間にある順序制約を利用する。

【 0 0 3 9 】

次に、本実施の形態における例外発生可能命令の投機的実行、副作用の投機的実行、補償コードの生成の手順を説明する。

プログラムは、プログラム中の演算子の依存関係を示す無閉路有向グラフ（Directed Acyclic Graph；以下、DAGと略す）として表現される。DAGの頂点は演算子を表し、DAGの辺（有向辺）は演算子間の依存関係を表す。依存関係は以下の3種類である。

データ依存：先行する演算子を書き込んだ値を後継する演算子を読み出すこと（フロー依存）、先行する演算子を読み出した場所に後継する演算子を書き込みを行うこと（逆依存）、及び先行する演算子を書き込んだ場所に後継する演算子を書き込みを行うこと（出力依存）を表す。

順序制約：先行する演算子の副作用が終わった後に後継する演算子の副作用が起きることを表す。

制御依存：先行する演算子による条件判定が真である場合にのみ後継する演算子が実行されることを表す。

【 0 0 4 0 】

例外発生可能命令の投機的実行は、プログラム中に存在する不要な順序制約をできるだけ緩和し、DAGの演算子間の並列性を抽出することを目的としている。DAG上で、経路上の演算を全て実行し終わるまでの時間が最も長い経路のことをグラフのクリティカルパス (critical path) と呼ぶ。したがって、DAGで表された演算を全て実行し終わるまでの時間の下限は、クリティカルパスの長さに対応することとなる。

本実施の形態では、DAGのクリティカルパス上にある頂点で、例外発生検出命令への順序制約、例外発生可能命令からの順序制約による辺を選び、例外発生可能命令の投機的実行を行うようにグラフを変形する。各変換は順序制約を除去するので、クリティカルパス長を短くすることができる。この操作を、クリティカルパス上の例外発生可能命令及び例外発生検出命令の順序制約がなくなるまで繰り返すことにより、投機的実行のオーバーヘッドを極小化し、効果を極大化することができる。

【 0 0 4 1 】

本実施の形態において、以上の処理を実現する四つ組中間コード最適化部 2 0 について説明する。

図 2 は、四つ組中間コード最適化部 2 0 の構成を説明するブロック図である。

図 2 を参照すると、四つ組中間コード最適化部 2 0 は、四つ組中間コードを DAG (Directed Acyclic Graph) に変換する DAG 生成部 2 1 と、DAG 編集部 2 2 と、DAG を四つ組中間コードに戻す四つ組中間コード再生部 2 3 とを備える。

DAG 生成部 2 1 は、四つ組中間コード生成部 1 0 により生成された四つ組中間コードを入力として受け取り、四つ組演算子間のデータ依存と順序制約を解析して、この解析結果を反映させた DAG を生成する。すなわち、DAG は、四つ組中間コードで表現されたプログラム中の各命令を頂点 (ノード) とし、命令間の順序制約を有向辺 (パス) で表している。

DAG編集部22は、DAG生成部21により生成されたDAGを入力として受け取り、例外発生可能命令を投機的に実行するようにDAGの編集を行う。

四つ組中間コード再生部23は、DAG編集部22により編集されたDAGを入力として受け取り、四つ組中間コード形式のプログラムに再度変換して出力する。

【0042】

図3は、DAG編集部22による編集処理の全体的な流れを説明するフローチャートである。

図3を参照すると、DAG編集部22は、まず、DAGで表現されたプログラム中のクリティカルパス上に存在する、例外による順序制約の緩和処理を行う（ステップ301）。この例外による順序制約の緩和によって、元のDAGに存在した例外発生可能命令は、例外の検出と例外ハンドラへの条件分岐とに分割される。そして、例外の検出から例外発生可能命令に対して新たに制御依存が追加される。

【0043】

次に、DAG編集部22は、例外による順序制約の緩和処理の施されたDAGに対して、例外ハンドラへの条件分岐のマージ処理を行う（ステップ302）。この処理により、連続した例外ハンドラへの条件分岐が、代表フラグを使用した一つの条件分岐にマージされる。

【0044】

次に、DAG編集部22は、プログラムを実行する計算機（以下、対象アーキテクチャと称す）が、プレディケート（predicate）付き命令が実行可能でかつ命令レベルの並列実行が可能かどうかを判断する（ステップ303）。

そして、対象アーキテクチャがプレディケート付き命令を並列実行する機能を持たない場合は、投機的に実行される例外に対する補償コードの生成処理を行う（ステップ304）。

一方、対象アーキテクチャがプレディケート付き命令を並列実行する機能を持つ場合は、ステップ301の例外による順序制約の緩和処理で生成された制御依存に対応してプレディケートの割付を行う（ステップ305）。

以上のようにして、補償コードの生成（ステップ 3 0 4）またはプレディケートの割付（ステップ 3 0 5）が終了した後、DAG 編集部 2 2 による処理を終了する。

【 0 0 4 5 】

図 4 は、図 3 のステップ 3 0 1 における例外による順序制約の緩和処理の詳細を説明するフローチャートである。

図 4 を参照すると、例外による順序制約の緩和処理において、DAG 編集部 2 2 は、まず、処理対象である DAG の頂点の中から、同じ例外発生条件を持つ例外発生可能命令 v と例外発生検出命令 t との対を探す（ステップ 4 0 1）。そして、検出された例外発生可能命令 v と当該例外発生検出命令 t とに対する順序制約がクリティカルパスを形成しているかどうか判断する（ステップ 4 0 2）。

【 0 0 4 6 】

この順序制約がクリティカルパスを形成している場合、すなわち、この順序制約がクリティカルパス上にある場合、DAG 編集部 2 2 は、元の、例外発生条件を検出して例外ハンドラへの分岐を行う命令 t を、例外条件の検出だけを行う命令 t' と、実際に例外ハンドラに条件分岐する命令 c とに分割する（ステップ 4 0 3）。

【 0 0 4 7 】

そして次に、DAG の頂点 t' から頂点 c へデータ依存の有向辺を張る（ステップ 4 0 4）。この辺は、命令 t' で検出された例外発生条件を利用して命令 c が分岐を行うことを示す。

また、頂点 t' から頂点 v へ制御依存の有向辺を張る（ステップ 4 0 5）。この辺は、命令 t' において例外が発生しないことが確認された場合にのみ命令 v が実行されることを示す。

また、順序制約で頂点 t に先行している頂点 p から頂点 c へ順序制約の有向辺を張る（ステップ 4 0 6）。この辺は、元のプログラムにおいて命令 t に先行していた全ての副作用が終了してから命令 c の条件分岐が実行されることを示す。

【 0 0 4 8 】

次に、DAG 編集部 2 2 は、命令 v が副作用を持っているか、すなわちメモリ

に対して書き込みを行うかどうかを判断する（ステップ407）。そして、命令 v が副作用を持っている場合、DAG編集部22は、DAGの順序制約で頂点 t に先行している頂点 p から頂点 v へ順序制約の有向辺を張る（ステップ408）。この辺は、変換が元のプログラムにおける副作用間の順序制約を保存することを示す。

【0049】

命令 v が副作用を持たない場合、及びステップ408を終了した後、DAG編集部22は、DAGの順序制約で頂点 v に後継している頂点 s へ、頂点 c から順序制約の有向辺を張る（ステップ409）。この辺は、元の命令 v に後継する順序制約は命令 c が実行された場合に保証されることを示す。

【0050】

次に、DAG編集部22は、DAGの順序制約で頂点 v に先行または後継している元々の辺（ステップ408で導入された辺は除く）を取り除く（ステップ410）。同様にして、順序制約で頂点 t に先行している辺を取り除く（ステップ411）。これらの処理により、例外発生可能命令に対する順序制約をクリティカルパス上から除去することが可能になる。

ステップ411の処理が終了すると、DAG編集部22は、ステップ401に戻り、クリティカルパス上の次の例外発生可能命令による順序制約を探す。そして、クリティカルパス上に順序制約が存在しないならば、例外による順序制約の緩和処理を終了する（ステップ402）。

【0051】

図5は、図3のステップ302における例外ハンドラへの条件分岐のマージ処理の詳細を説明するフローチャートである。

図5を参照すると、例外ハンドラへの条件分岐のマージ処理において、DAG編集部22は、まず、例外による順序制約の緩和処理の済んだDAGのうち、順序制約の辺によって結ばれた例外ハンドラへの条件分岐の組 c_1 、 c_2 を探す（ステップ501）。そして、そのような条件分岐の組 c_1 、 c_2 が存在するかどうかを判断する（ステップ502）。

【0052】

条件分岐の組 c_1 、 c_2 が存在する場合、DAG 編集部 22 は、DAG の順序制約で条件分岐 c_1 に後継している頂点 s の制御依存に対して、条件分岐 c_1 のガード条件を論理積によって追加する（ステップ 503）。追加された制御依存は、条件分岐 c_1 がチェックしていた条件と元々命令 s が実行される条件の両方が成立した場合にのみ命令 s が実行されることを示す。

【0053】

次に、DAG 編集部 22 は、DAG の条件分岐 c_1 から頂点 s に対する順序制約を取り除き（ステップ 504）、条件分岐 c_1 と条件分岐 c_2 とをマージする（ステップ 505）。マージされた条件分岐 c_2 では、元の条件分岐 c_1 と条件分岐 c_2 の条件の両方がテストされる。

ステップ 505 の処理が終了すると、DAG 編集部 22 は、ステップ 501 に戻り、順序制約の辺によって結ばれた例外ハンドラへの条件分岐の組 c_1 、 c_2 を探す。そして、そのような条件分岐の組 c_1 、 c_2 が存在しないならば、例外ハンドラへの条件分岐のマージ処理を終了する（ステップ 502）。

【0054】

図 6 は、図 3 のステップ 304 における投機的に実行される例外に対する補償コードの生成処理の詳細を説明するフローチャートである。

図 6 を参照すると、投機的に実行される例外に対する補償コードの生成処理において、DAG 編集部 22 は、まず、コードスケジューリング最適化を行い、DAG 上において例外発生可能命令と副作用の間の順序を決定する（ステップ 601）。このコードスケジューリング最適化では、演算子の遅延時間と対象計算機の資源の使用状況を考慮して、DAG の頂点に対して最適な順序付けを行う。

【0055】

次に、DAG 編集部 22 は、順序が決定した DAG の頂点のうち、例外による順序制約の緩和処理（図 3 のステップ 301 及び図 4 参照）において投機的実行の対象となった例外発生検出命令 t に対して、元のプログラム中で推移的に先行していた頂点 p を探す（ステップ 602）。

【0056】

次に、DAG 編集部 22 は、ステップ 602 により得られた DAG の頂点の組

t、pのうち、ステップ601のコードスケジューリング最適化によって与えられた順序によって元々のプログラムの順序制約が守られない組が存在するかどうかを調べる（ステップ603）。

【0057】

元々のプログラムの順序制約が守られない頂点tと頂点pの組が存在する場合、次にDAG編集部22は、条件分岐として生成される命令tの分岐先に、命令pを補償コードとして生成する（ステップ604）。DAGの頂点tに対して先行する頂点pが複数存在する場合は、各頂点pについて、元のプログラムの順序制約を満たす順に、命令tの分岐先に補償コードを生成する。

ステップ604の処理が終了すると、DAG編集部22は、ステップ602に戻り、DAGにおいて次の順序が満たされない頂点tと頂点pの組を探す。そして、順序が満たされない頂点tと頂点pの組が存在しないならば、投機的に実行される例外に対する補償コードの生成を終了する（ステップ603）。

【0058】

以上のようにしてDAG編集部22によるDAGの編集が終了した後、四つ組中間コード再生部23が編集の済んだDAGを四つ組中間コードに戻す。そして、機械語コード生成部30が、最適化の済んだ当該四つ組中間コードを、対象アーキテクチャに対応した機械語コードに変換してコンパイルを終了する。

【0059】

次に、Javaプログラムの中間コードを最適化する具体的な動作例について説明する。

例として、図7（A）に示すJavaプログラムの中間コードを最適化する場合を考える。NullPointerExceptionのチェックがソフトウェアによって行われるプラットフォームを仮定すると、このプログラムに対応する四つ組中間コードは図7（B）に示すように表される。

図7において、NULLは配列のNullPointerExceptionを検出する操作、LENGTHは配列の長さを得る操作、SIZEはArrayIndexOutOfBoundsExceptionを検出する操作を表す。四つ組中間コード最適化部20のDAG生成部21により、このプログラムの持つ制約をDAGで表現した様子を図8に示す。

【 0 0 6 0 】

次に、四つ組中間コード最適化部 2 0 の D A G 編集部 2 2 が、図 8 に示す D A G を編集する。

図 8 に示す D A G において、簡単のために全ての演算のコストを 1 とすると、このグラフのクリティカルパス長は 8 サイクルである。SIZE 命令と NULL 命令は制御を変更するので、SIZE 命令から NULL 命令の間に順序制約がある。このため NULL 命令に依存する LENGTH、SIZE、LOAD が SIZE 命令に依存を持ち、データ依存による本来のクリティカルパス長を伸ばしている。

【 0 0 6 1 】

まず、例外による順序制約の緩和処理（図 3 のステップ 3 0 1 及び図 4 参照）において、NULL 命令を投機的に実行することにより、制約を緩和する。元々の NULL 命令の動作は、例外発生を検出と例外ハンドラへの分岐からなる。これを、例外発生条件の検出（同じ NULL 命令で表す）と、例外処理ルーチンへの分岐（CHECK 命令）とに分割する。

例外発生検出命令（NULL）は、例外発生条件を検出すると、基本ブロック内で例外が発生したことを示す代表フラグを設定する。例外処理ルーチンへの分岐命令（CHECK）は、代表フラグを調べ、もし例外が発生していれば、元のプログラムで最初に発生するはずの例外ハンドラへ分岐する。

以上の処理で、代表フラグを用いることにより、複数の例外の発生を 1 命令で検出することができる。例外発生可能命令である LENGTH 命令は、例外発生検出命令（NULL）によってガードされる。すなわち、NULL 命令が例外発生条件を検出すると、LENGTH 命令は実行されない。元の NULL 命令から例外ハンドラへの分岐を分離することにより、NULL 命令に先行する SIZE 命令による順序制約を外すことができる。

例外発生可能命令の値を使用している他の演算については、例外発生可能命令や副作用でない限りガードする必要はない。なぜなら、例外ハンドラで観測できるのは副作用と例外の発生だけだからである。

【 0 0 6 2 】

以上の変形を、下の SIZE 命令にも同様に適用し、例外ハンドラへの条件分岐の

マージ処理（図 3 のステップ 3 0 2 及び図 5 参照）を行うことによって得られる DAG を図 9 に示す。

図 8 と図 9 の DAG を比較すると、図 8 の DAG ではクリティカルパス長が 8 サイクルであったが、上記の処理で CHECK 命令を一つ追加することにより、図 9 に示すように、クリティカルパス長が 5 サイクルに短縮された。

図 8、図 9 において、データ依存は実線矢印、順序制約は一点鎖線矢印、制御依存は破線矢印で表す。後述の図 1 1、図 1 2 においても同様である。

【 0 0 6 3 】

次に、副作用を伴う例外発生可能命令の投機的実行について具体例を挙げて説明する。副作用を伴う例外発生可能命令の投機的実行を行う場合は、直前の例外発生検出命令だけでなく、先行する全ての例外発生検出命令の条件でガードする必要がある。

例として、図 1 0 に示す J a v a プログラムの中間コードを最適化する場合について考える。NullPointerException のチェックがソフトウェアによって行われるプラットフォームを仮定すると、このプログラムに対応する四つ組中間コードは図 1 0 （B）に示すように表される。四つ組中間コード最適化部 2 0 の DAG 生成部 2 1 により、このプログラムの持つ制約を DAG で表現した様子を図 1 1 に示す。

【 0 0 6 4 】

次に、四つ組中間コード最適化部 2 0 の DAG 編集部 2 2 が、図 1 1 に示す DAG を編集する。

図 1 1 に示す DAG において、全ての演算のコストを 1 とすると、このグラフのクリティカルパス長は 1 0 サイクルである。3 番目の SIZE 命令を投機的に実行する場合、STORE 命令による副作用は、当該 STORE 命令に先行する全ての例外発生検出命令における条件の積によってガードされなければならない。

【 0 0 6 5 】

図 1 1 に示す DAG に対し、例外による順序制約の緩和処理（図 3 のステップ 3 0 1 及び図 4 参照）及び例外ハンドラへの条件分岐のマージ処理（図 3 のステップ 3 0 2 及び図 5 参照）を行うことによって得られる DAG を図 1 2 に示す。

図11と図12のDAGを比較すると、図11のDAGではクリティカルパス長が10サイクルであったが、上記の処理でCHECK命令を追加することにより、図12に示すように、クリティカルパス長が6サイクルに短縮された。

【0066】

次に、例外発生が検出された場合の補償コードの生成について、具体的な動作例を挙げて説明する。

上述したように、例外が発生した場合には、元のプログラムで例外発生可能命令より前に実行されている副作用と例外の検出を実行しなければならない。IA-64のようなプレディケート付き命令を実行可能なアーキテクチャでは、例外発生検出命令から例外発生可能命令及び副作用への制御依存をプレディケートで実装すれば、補償コードは不要である。この場合、代表フラグのテストでは、基本ブロック内で起った複数の例外のうち、元のプログラム中の最初に存在する例外発生検出命令の例外ハンドラに分岐すればよい。

【0067】

一方、プレディケート付き命令を実行できないアーキテクチャでは、条件分岐によって例外発生検出命令によるガードが実装される。そのため、例外発生を検出した場合の分岐先に、必要な副作用の実行と例外検出を行うための補償コードが必要となる（図3のステップ304及び図6参照）。補償コードは、コードスケジューリング最適化が終了して、演算間の順序が決定した後を求める。

【0068】

図13は、所定のコード（図13（a））に対して、必要な補償コードを補った様子（図13（b））を説明する図である。また、図14は、図13に示すコードの順序制約を示す図である。

図13において、下図でtestは例外発生検出命令、testの間のアルファベットは副作用のある演算を表すとする。副作用と例外発生検出命令の間には、上述した推移的な制御依存がある。すなわち、副作用 AB_1 はtest（A）とtest（B）の前に実行することはできない。

【0069】

図13（a）に示す例外発生検出命令と副作用の順番で命令が実行されるので

あれば、補償コードは不要である。すなわち、例外発生検出命令から、ガードされている演算を全てスキップするように条件分岐を生成すればよい。

これに対し、コードスケジューリング最適化によって、図 1 3 (b) に示すように例外発生検出命令と副作用が配置された場合、例外発生検出命令及び副作用の関係は図 1 3 (a) の場合のように単純ではない。すなわち、例外発生が検出されたパスで必要な補償コードは、図 1 4 (a) に示す元のコードにおける例外及び副作用の順序制約を溯った場合には実行されていなければならない。しかし、かかる補償コードは、実際には例外が検出されるまで実行されておらず、例外発生検出命令の分岐から復帰した後も実行されない。

【 0 0 7 0 】

例えば、test (B) で例外が検出された場合、この時点ではtest (A)、 A_0 が実行されていない。図 1 3 (a) に示す元のコードではtest (A) で例外が発生しなければ副作用 A_0 が実行されているので、図 1 3 (b) に示すように、test (A) で A_0 をガードしたコードを挿入する。同様に、test (C) で例外が検出された場合、この時点では副作用 AB_1 がまだ実行されていないので、図 1 3 (b) に示すように、 AB_1 を補償コードとして生成する。

【 0 0 7 1 】

もし、test (A) とtest (B) における例外が例外ハンドラから見て区別できない場合は、補償コードをさらに少なくすることができる。例えば、図 1 3 (a) に示す元のコードにおいて、 A_0 の位置に副作用がなく、test (A) とtest (B) が同じ種類の例外であれば、順序制約は図 1 4 (b) のようになる。この場合、図 1 3 (b) に示した順序でコードが配置されても、副作用 A_0 が存在しないので、test (B) の補償コードによりtest (A) における例外の発生を検出する必要はない。

【 0 0 7 2 】

次に、本実施の形態を用いた、数種類の対象アーキテクチャ（プログラムを実行する計算機）に対応したコード生成の例について説明する。

まず、x 8 6 アーキテクチャ（米インテル社製のプロセッサにおける 8 0 8 6、8 0 2 8 6、3 8 6 アーキテクチャ）を対象としたコード生成例を説明する。

X 8 6 アーキテクチャは、ユーザー定義のフラグレジスタ及びプレディケートを持たない。X 8 6 アーキテクチャ上の主要なオペレーティングシステムである W i n 3 2、O S / 2、L i n u x では、仮想メモリ上の 0 番地から始まるページには読み出し及び書き込み保護が設定されているので、J a v a 言語の N u l l P o i n t e r E x c e p t i o n の判定にはハードウェアによるアクセス違反例外の例外ハンドラで検出することが可能である。したがって、例外発生可能命令を投機実行しない場合は、N u l l P o i n t e r E x c e p t i o n の検出命令に対しては実際にコード生成をしなくてよい。

【 0 0 7 3 】

しかし、ソフトウェアによる投機的実行を行う場合は、例外発生可能命令をガードする必要があるので、N u l l P o i n t e r E x c e p t i o n の検出命令に対応する命令を明示的に生成しなければならない。W i n 3 2 プラットフォームを対象とした場合には、J a v a バイトコードの i a l o a d 命令の投機的実行に必要な基本操作は、図 1 5 に示すように実装される。

図 1 5 において、a r r h は配列オブジェクトのヘッダを格納するレジスタ、i d x は配列アクセスの添字の値を格納するレジスタである。e H a n d l e r は元々の例外ハンドラの先頭を示す。f l H a n d l e r は補償コードの実行及び代表フラグの設定を行うコードの先頭を示す。また、s e l H a n d l e r は最初に発生した例外を調べて元の e H a n d l e r に分岐するコードの先頭を示す。

【 0 0 7 4 】

i a l o a d 命令の投機的実行に必要な基本操作が図 1 5 に示すように実装されることは、投機的に実行されるメモリアクセス命令がデータ依存及び副作用だけを考慮した D A G のクリティカルパス上にあった場合に、投機的実行によってクリティカルパスの長さを伸ばしてしまう可能性があることを意味している。

【 0 0 7 5 】

この問題を回避するため、以下のような手法を取る。

すなわち、データ依存と副作用だけを考慮して得られる D A G について、各頂点の自由度 (slackness) を計算する。自由度とは、当該頂点を含む経路に対して、全体のクリティカルパス長を伸ばすことなく、後何サイクルの演算を挿入す

ることができるかを表す数字である。投機的実行を行う例外発生可能命令を選択する場合に、自由度が条件分岐命令のコストよりも小さければ、その例外発生可能命令の投機的実行を行わない。

なお、各頂点の自由度は図 1 6 に示すようにして計算される。

【 0 0 7 6 】

次に、Power PC アーキテクチャ（米国 IBM 社、米国アップルコンピュータ社、米国モトローラ社が開発した CPU である Power PC におけるアーキテクチャ）を対象としたコード生成例を説明する。

Power PC アーキテクチャは、分岐条件を記憶する専用のコンディションレジスタを持つ。上述した X 8 6 アーキテクチャのコード生成例と同様に、本手法を適用することによって、コードスケジューラが例外発生検出命令の比較演算と通常の演算における計算との並列性を利用することが可能になる。Power PC アーキテクチャでは、比較演算と分岐命令の間に 3 サイクル以上のレイテンシを置けば、分岐命令は命令フェッチユニットで処理される。このため見かけ上、分岐命令のオーバーヘッドが 0 になる（0 サイクル分岐）。したがって、例外発生可能命令及び例外発生検出命令の並列性を抽出することで、クリティカルパス長を伸ばすことなく演算パイプラインの充填率を上げることが可能になる。AIX（米国 IBM 社の UNIX）プラットフォームを対象とした場合、Java バイトコードの iaload 命令の投機的実行に必要な基本操作は図 1 7 に示すように実装される。

図 1 7 において、arrh、flHandler、selHandler、idx の意味は上述した x 8 6 アーキテクチャにおける図 1 5 の場合と同様である。len は、配列の長さを格納するレジスタである。

【 0 0 7 7 】

次に、IA-64 アーキテクチャを対象としたコード生成例を説明する。

IA-64 アーキテクチャは、プレディケート付き命令フォーマットを持つ。これにより、命令がプレディケートでガードされており、プレディケートの真偽値にしたがって命令が実行される。したがって、先に述べたようにガード付きで表現された DAG をプレディケートで実装することによって、例外が発生した場

合の補償コードを生成せずに済ませることができる。プレディケートで実装されたコードに対しては、コードスケジューリング及びコード生成を直接適用することができる。WIN64プラットフォームを対象とした場合、Javaバイトコードのiaload命令の投機的実行に必要な基本操作は図18に示すように実装される。

図18において、arrh、idx、eHandler、selHandlerの意味は上述したx86アーキテクチャにおける図15の場合と、lenの意味は上述したPowerPCアーキテクチャにおける図17の場合と、それぞれ同様である。プレディケートの論理積は、並列比較を用いることによって、余分な命令オーバーヘッドを伴わずに実現できる。

【0078】

以上のように、従来の技術ではハードウェアによるサポートが必要であった例外発生可能命令の投機的実行が、本実施の形態によってソフトウェアで実装可能となる。これにより、Javaのような例外発生可能命令の順序制約が厳密である言語において、コードスケジューラによる命令レベルの並列性の抽出を妨げる順序制約を解消することが可能になった。

また上記のように、本実施の形態は、プレディケートを持たないプロセッサに実装した場合でも、例外ハンドラへの条件分岐命令を基本ブロックあたり2命令追加するだけで、例外発生検出命令の付加的なオーバーヘッド無しに例外発生可能命令の投機的実行を実現できる。

【0079】

【発明の効果】

以上説明したように、本発明によれば、例外発生可能命令の他の命令への先行制約をソフトウェア的に緩和することによって、例外発生可能命令の投機的実行を実現でき、例外発生可能命令を含むプログラムの命令レベル並列性を効果的に得ることが可能となる。

【図面の簡単な説明】

【図1】 本発明の実施の形態におけるコンパイラの構成を説明する図である。

【図 2】 本実施の形態における四つ組中間コード最適化部の構成を説明するブロック図である。

【図 3】 本実施の形態における D A G 編集部による編集処理の全体的な流れを説明するフローチャートである。

【図 4】 図 3 のステップ 3 0 1 における例外による順序制約の緩和処理の詳細を説明するフローチャートである。

【図 5】 図 3 のステップ 3 0 2 における例外ハンドラへの条件分岐のマージ処理の詳細を説明するフローチャートである。

【図 6】 図 3 のステップ 3 0 4 における投機的に実行される例外に対する補償コードの生成処理の詳細を説明するフローチャートである。

【図 7】 本実施の形態の処理対象であるプログラムの中間コード及びその四つ組中間コードの例を示す図である。

【図 8】 図 7 のプログラムの持つ制約を D A G で表現した図である。

【図 9】 図 8 の D A G に対して編集を行った状態を示す図である。

【図 1 0】 本実施の形態の処理対象であるプログラムの中間コード及びその四つ組中間コードの他の例を示す図である。

【図 1 1】 図 1 0 のプログラムの持つ制約を D A G で表現した図である。

【図 1 2】 図 1 1 の D A G に対して編集を行った状態を示す図である。

【図 1 3】 所定のコードに対して、必要な補償コードを補った様子を説明する図である。

【図 1 4】 図 1 3 に示すコードの順序制約を示す図である。

【図 1 5】 x 8 6 アーキテクチャにおいて、J a v a バイトコードの i a l o a d 命令の投機的実行を行うために必要な基本操作図の実装例を示す図である。

【図 1 6】 データ依存と副作用だけを考慮して得られる D A G について、各頂点の自由度を計算する手法を説明する図である。

【図 1 7】 P o w e r P C アーキテクチャにおいて、J a v a バイトコードの i a l o a d 命令の投機的実行を行うために必要な基本操作図の実装例を示す図である。

【図 1 8】 I A - 6 4 アーキテクチャにおいて、J a v a バイトコードの

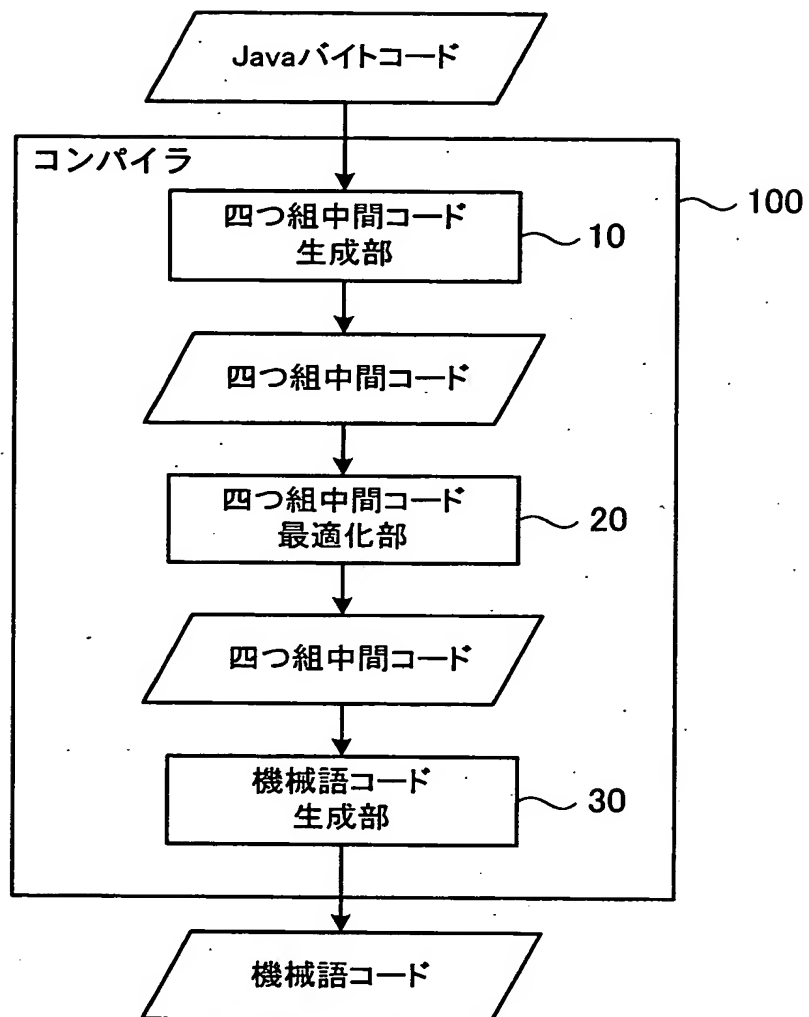
iaload命令の投機的実行を行うために必要な基本操作図の実装例を示す図である。

【符号の説明】

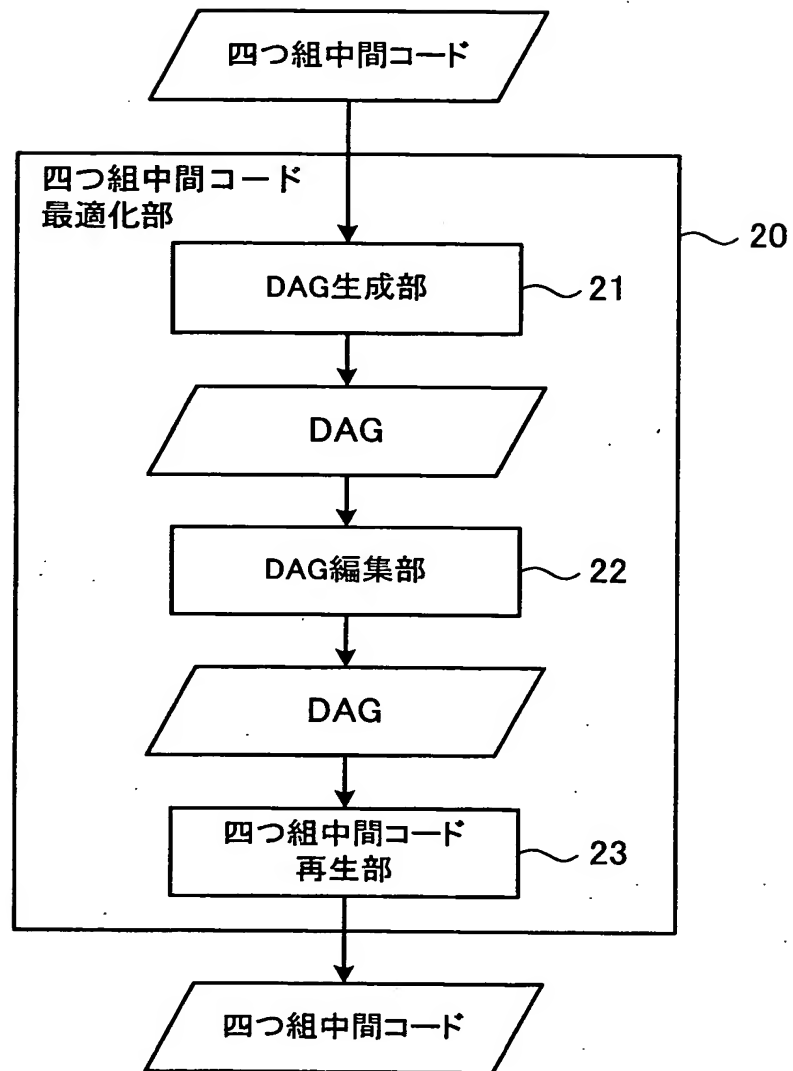
1 0 …四つ組中間コード生成部、2 0 …四つ組中間コード最適化部、2 1 …D A G生成部、2 2 …D A G編集部、2 3 …四つ組中間コード再生部、3 0 …機械語コード生成部

【書類名】 図面

【図 1】

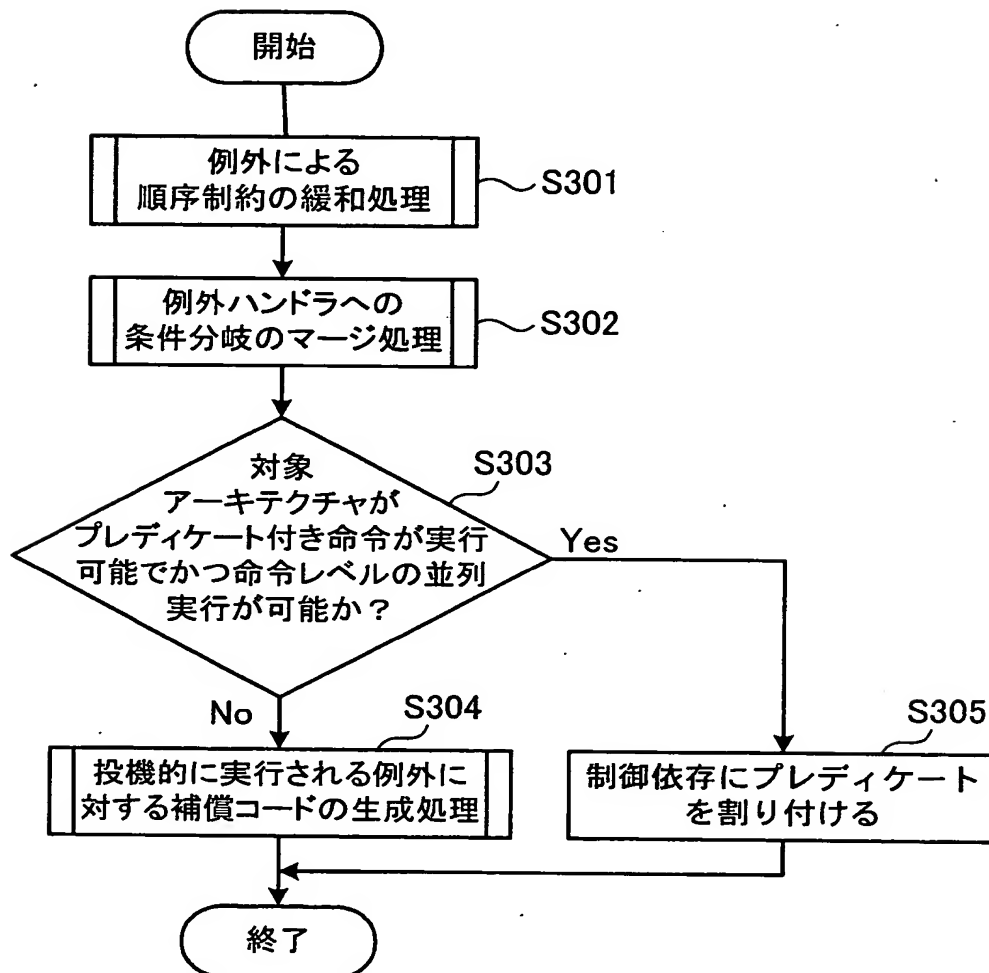


【図 2】



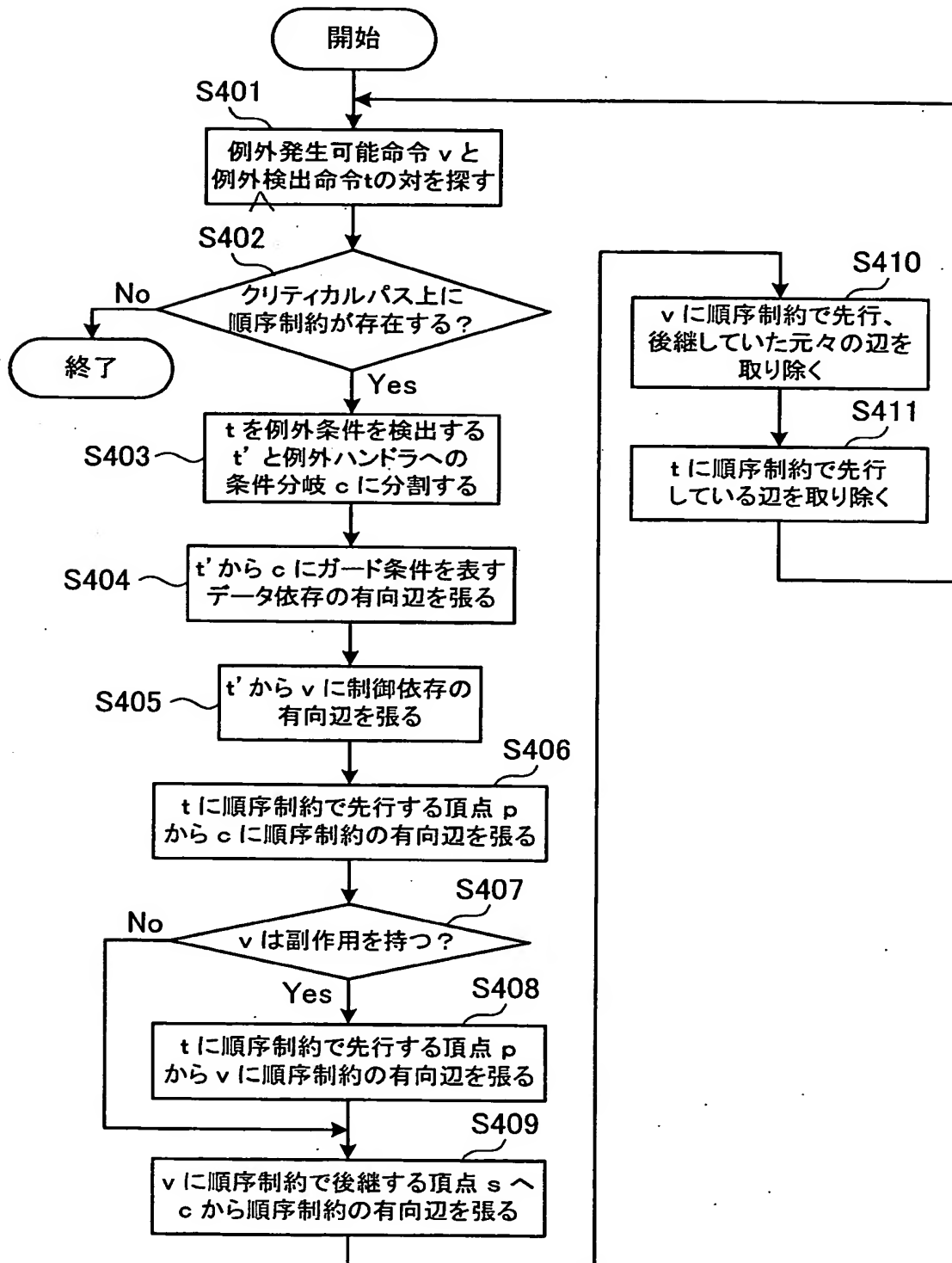
【図 3】

DAG編集部による
例外発生可能命令の投機的実行



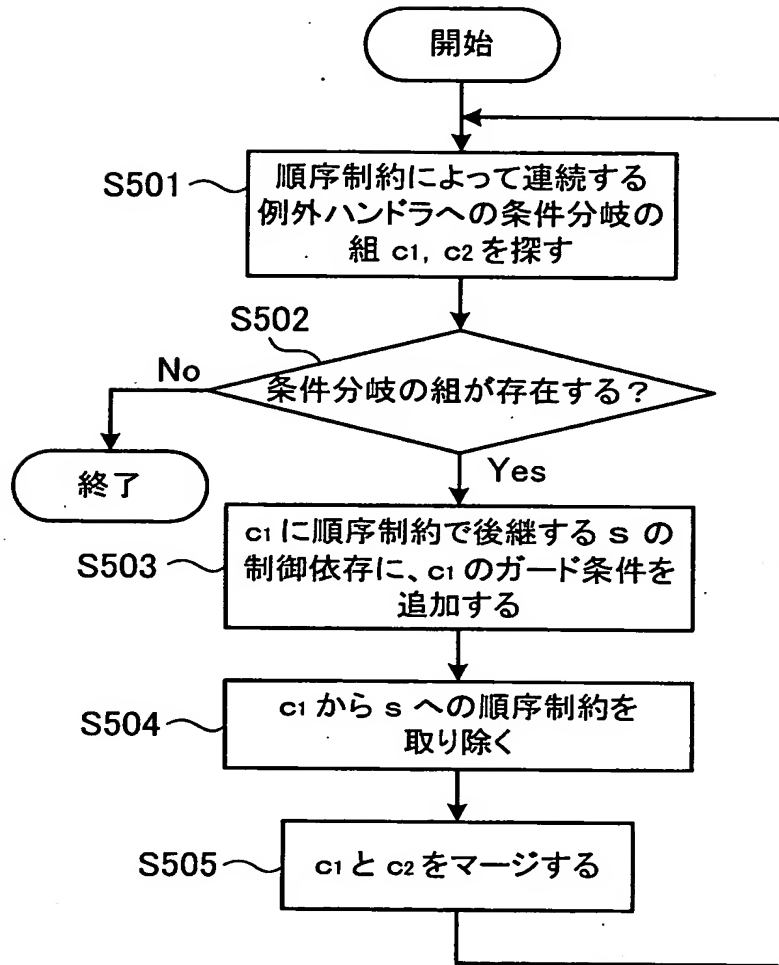
【図 4】

例外による順序制約の緩和処理



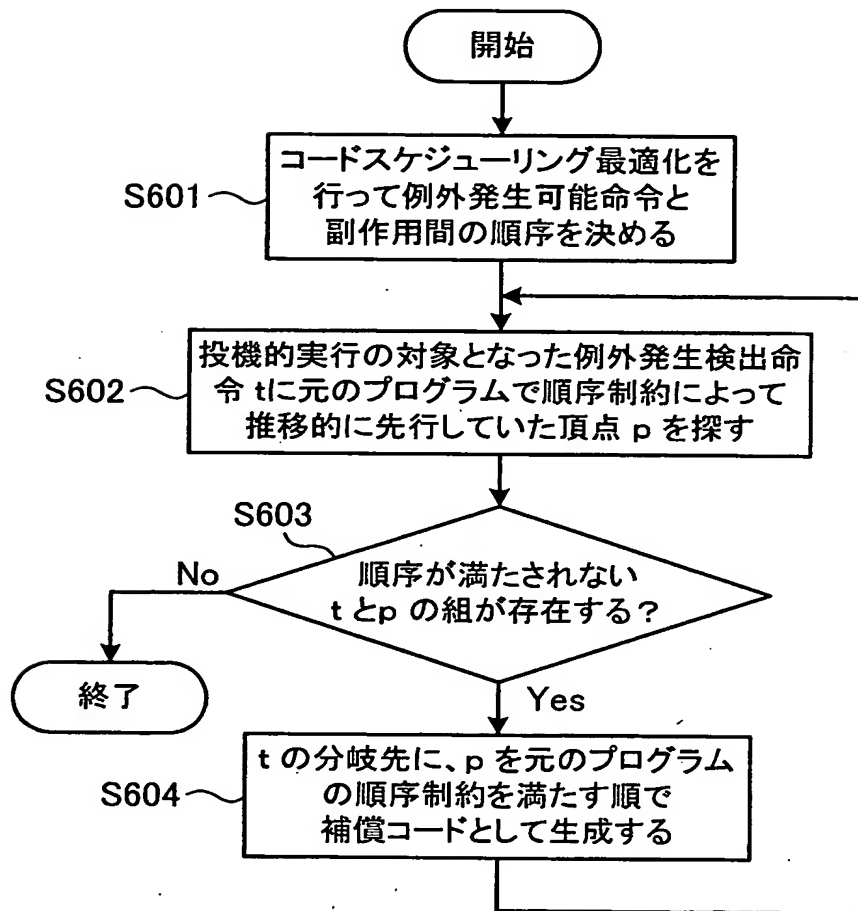
【図 5】

例外ハンドラへの条件分岐のマージ処理



【図 6】

投機的に実行される例外に対する
補償コードの生成処理



【図 7】

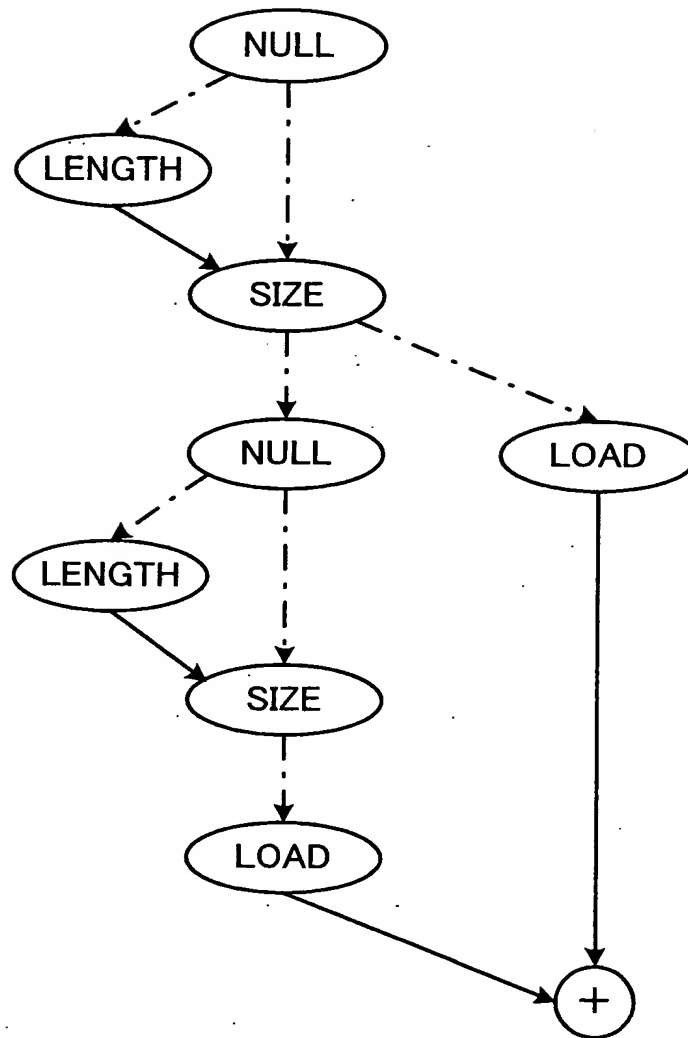
(A)

```
double test (double a[], double b[], int i, int j) {
    return a[i] + b[j];
}
```

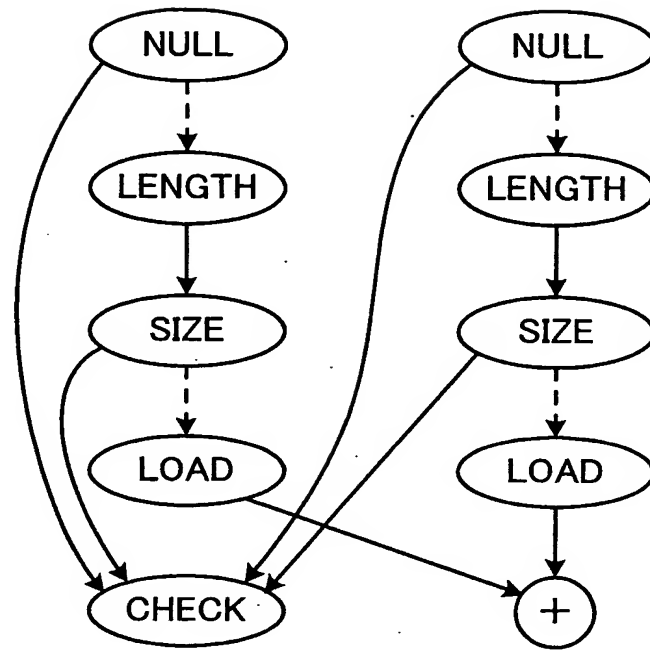
(B)

NULL		a
LENGTH	t=	a
SIZE		t, j
LOAD	x=	a, j
NULL		b
LENGTH	t=	b
SIZE		t, k
LOAD	y=	b, k
ADD	z=	x, y

【図 8】



【図 9】



【図 1 0】

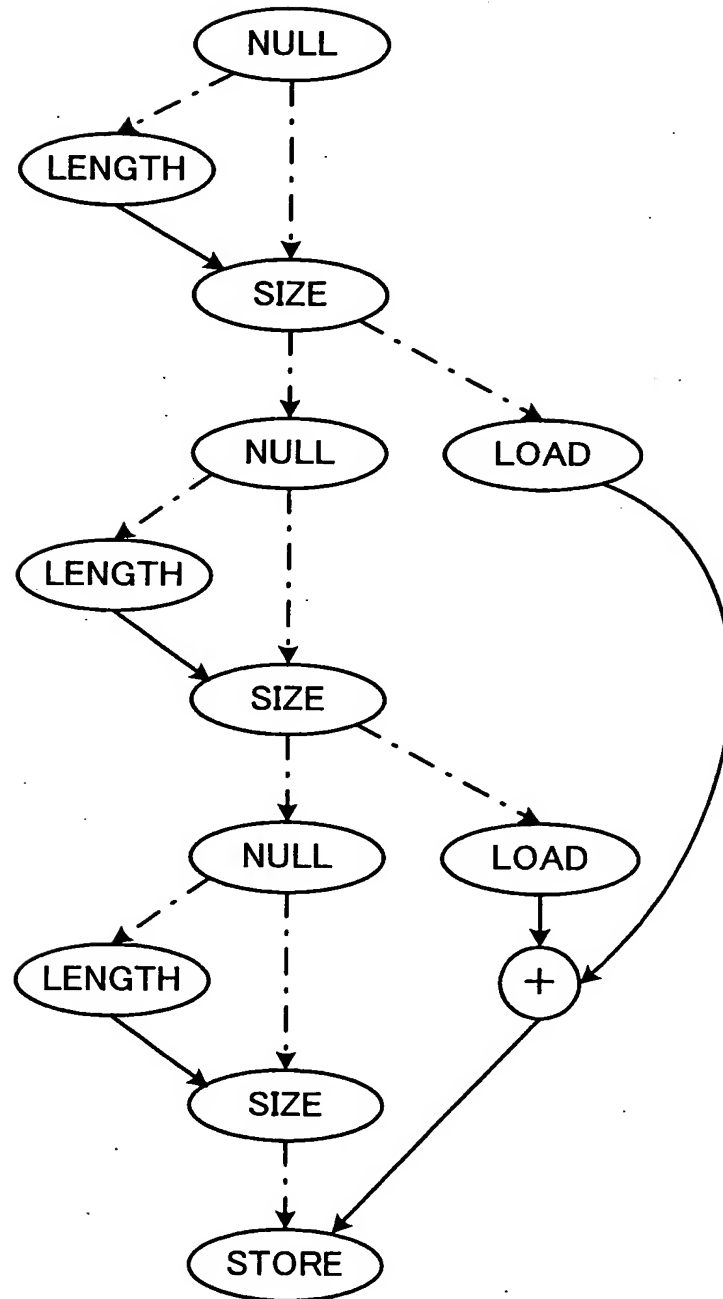
(A)

```
void    test    (double a[], double b[], double c[],
                  int i, int j, int k)  {
    a[i] = b[j] + c[k];
}
```

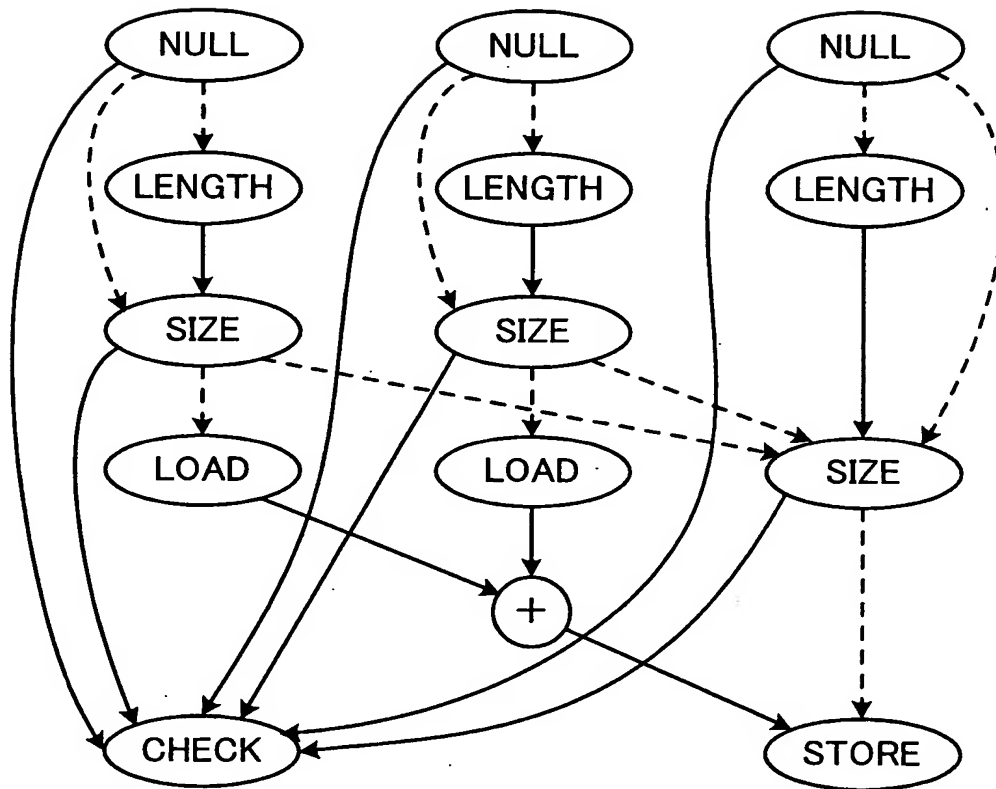
(B)

NULL		b
LENGTH	t=	b
SIZE		t, j
LOAD	x=	b, j
NULL		c
LENGTH	t=	c
SIZE		t, k
LOAD	y=	c, k
ADD	z=	x, y
NULL		a
LENGTH	t=	a
SIZE		t, i
STORE		a, i, z

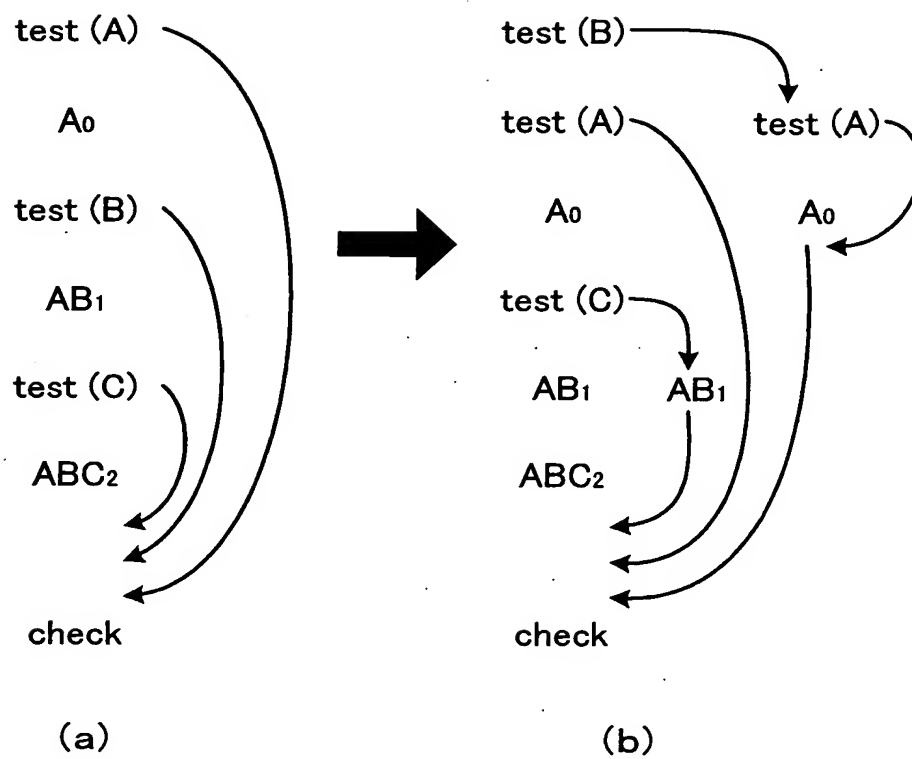
【図 1 1】



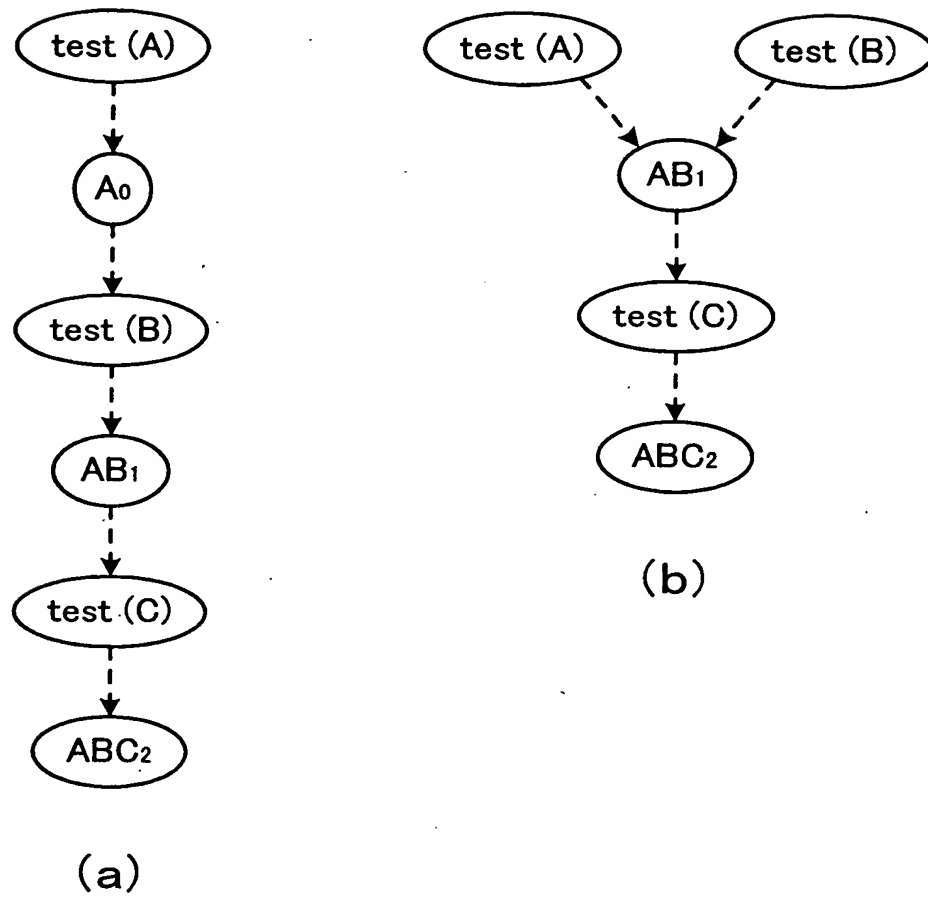
【図12】



【図 1 3】



【図 1 4】



【図 1 5】

	NullPointerException	ArrayIndexOutOf BoundsException	代表フラグのチェック
通常実行	なし	cmp idx, [arrh] jae eHandler	なし
投機的実行	cmp arrh, 00h jeq flHandler	cmp idx, [arrh] jae flHandler	cmp flag, 00h jne selHandler

【図 1 6】

入力

- V : 四つ組の演算子に対応する頂点の集合
- E : 演算子間の依存関係に対応する辺の集合
- $G(V, E)$: プログラムを表す DAG
- $\tau \in V$: 先行する演算子を持たない全ての演算子に先行する仮想的な頂点 (トップ)
- $\perp \in V$: 後継する演算子を持たない全ての演算子に後継する仮想的な頂点 (ボトム)
- $cy(v)$: 頂点 v の演算を実行するために必要なマシンサイクル数。
- $cp(G)$: 例外による順序制約を無視したときのプログラムを表す DAG の critical path 長

手続き (頂点の自由度の計算)

- $lb(v)$: 例外による順序制約を無視したときの頂点 v のボトムからのレベル
1. 後継する頂点がない場合は $lb(v)=0$.
 2. 後継する頂点がある場合は、 v に例外による順序制約以外の辺で後継する頂点の集合を $s \in succ(v)$ と表すと、
$$lb(v) = \max (lb(s)) + cy(v).$$
- $lt(v)$: 例外による順序制約を無視したときの頂点 v のトップからのレベル
1. 先行する頂点がない場合は $lt(v)=0$.
 2. 先行する頂点がある場合は、 v に例外による順序制約以外の辺で先行する頂点の集合を $p \in pred(v)$ と表すと、
$$lt(v) = \max (lt(p) + cy(v)).$$
- $sl(v)$: 例外による順序制約を無視したときの頂点 v の自由度
- $$sl(v) = cp(G) - lt(v) - lb(v).$$

【図 1 7】

	NullPointerException	ArrayIndexOutOf BoundsException	代表フラグのチェック
通常実行	tlei arrh, 00h	lw len, [arrh] tllc idx, len	なし
投機的実行	cmp arrh, 00h beq flHandler	lw len, [arrh] cmpl idx, len bgt flHandler	bne cr2, selHandler

【図 1 8】

	NullPointerException	ArrayIndexOutOf BoundsException	代表フラグのチェック
通常実行	dmp.ne p1, p2 - 00h, arrh (p1) br eHandler	ld8 len = [arrh] cmp.ltu p1, p2 - idx, len (p1) br eHandler	なし
投機的実行	cmp.ne.and p1,p2 = 00h, arrh	ld8 len = [arrh] cmp.ltu.and p1, p2 = idx, len	(p1) br selHandler

【書類名】 要約書

【要約】

【課題】 例外発生可能命令の他の命令への先行制約をソフトウェア的に緩和し、例外発生可能命令を含むプログラムの命令レベル並列性を効果的に得ることを可能とする。

【解決手段】 コンパイラにおいて、処理対象のプログラムの四つ組中間コードを解析し、DAGを生成するDAG生成部21と、生成されたDAGを編集し、例外による演算子の順序制約を緩和するDAG編集部22と、編集後のDAGの構造を反映させた四つ組中間コードを生成する四つ組中間コード再生部23とを備える。そして、DAG中から、例外発生可能命令と例外発生検出命令とを検出し、検出された例外発生検出命令を、例外の発生条件を検出する第1の命令と、例外処理へ処理を条件分岐させる第2の命令とに分割し、第1の命令に対して、例外の発生条件を検出した場合に第2の命令に移行し、例外発生条件を検出しなかった場合に例外発生可能命令に移行するように依存関係を設定する。

【選択図】 図2

認定・付加情報

特許出願の番号	特願 2 0 0 0 - 3 3 1 4 2 7
受付番号	5 0 0 0 1 4 0 4 7 5 8
書類名	特許願
担当官	塩崎 博子 1 6 0 6
作成日	平成 1 2 年 1 2 月 1 3 日

<認定情報・付加情報>

【特許出願人】

【識別番号】	390009531
【住所又は居所】	アメリカ合衆国 1 0 5 0 4、ニューヨーク州 アーモンク (番地なし)
【氏名又は名称】	インターナショナル・ビジネス・マシーンズ・コーポレーション

【代理人】

【識別番号】	100086243
【住所又は居所】	神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ビー・エム株式会社 大和事業所内
【氏名又は名称】	坂口 博

【代理人】

【識別番号】	100091568
【住所又は居所】	神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ビー・エム株式会社 大和事業所内
【氏名又は名称】	市位 嘉宏

【代理人】

【識別番号】	100106699
【住所又は居所】	神奈川県大和市下鶴間 1 6 2 3 番 1 4 日本アイ・ビー・エム株式会社大和事業所内
【氏名又は名称】	渡部 弘道

【復代理人】

【識別番号】	100104880
【住所又は居所】	東京都港区赤坂 5 - 4 - 1 1 山口建設第 2 ビル 6 F セリオ国際特許事務所
【氏名又は名称】	古部 次郎

【選任した復代理人】

【識別番号】	100100077
--------	-----------

次頁有

認定・付加情報（続き）

【住所又は居所】	東京都港区赤坂 5 - 4 - 1 1 山口建設第 2 ビル 6 F セリオ国際特許事務所
【氏名又は名称】	大場 充

出 願 人 履 歴 情 報

識別番号 [390009531]

1. 変更年月日 2000年 5月16日

[変更理由] 名称変更

住 所 アメリカ合衆国10504、ニューヨーク州 アーモンク (番地なし)

氏 名 インターナショナル・ビジネス・マシーンズ・コーポレーション